|  |  |  |
| --- | --- | --- |
| **Project report** | | Academic year:  **2020/21** |
| Subject:  **Algorithms and data structures** | |
| Project name:  **Dynamic programing** | | Subject takes place at:  **N/A** |
| Faculty, field of study, semester:  **FC, AI, II** | Full name:  **Jan Gruszczyński** | Grade: |
|

**Exercise 1**

Solving 0-1 knapsack problem employing brute force method, greedy ratio algorithm and dynamic programing.

At each measuring point, problem was solved 30 times on consequently new generated lists of elements, then an average was taken as representative of obtained values.

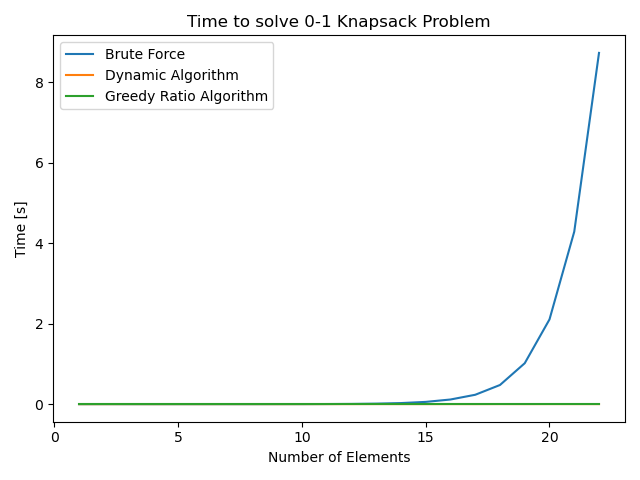


Figure 1: Efficiency comparison of said algorithms. Weights: (1-25), Values: (10-100), Max knapsack weight: 50.

Clearly, brute force is the least efficient method, as it checks every possible solution, for every n items that can be chosen, there are 2n combinations of items that can be packed into knapsack. Thus brute force computational complexity is O(n\*2n)=O(2n). Thesis is supported by the graph, showing that computational complexity of brute force grows exponentially, so it can be used only for small instances.

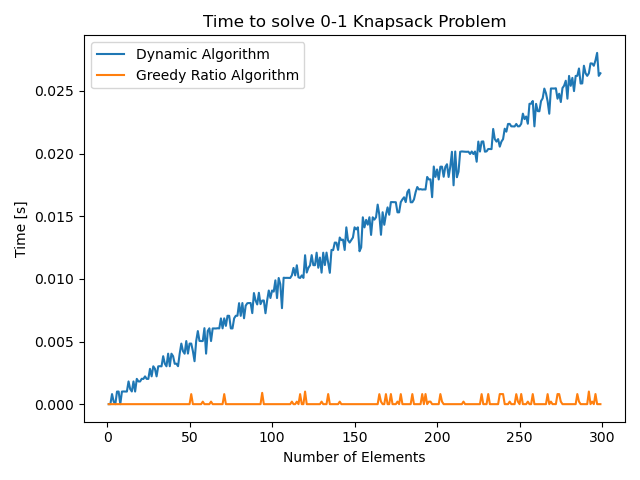


Figure 2: Efficiency comparison of said algorithms. Weights: (1-25), Values: (1-30), Max knapsack weight: 50. Brute force omitted.

Dynamic algorithm divides knapsack problem into smaller subproblems, while recording their results in two dimensional array. This allows the algorithm to avoid solving unnecessary overlapping problems (like in the brute force approach). The algorithm divides all the subsets of the first k items that fit in the knapsack of max weight p into two categories: subsets that do not include the k’th item and subsets that include the k’th item. Thus dynamic algorithm computational complexity is O(number\_of\_elements\*max\_knapsack\_weight). Same for the memory complexity. Thesis is supported by the graph, showing that computational complexity of dynamic algorithm grows linearly (as max\_knapsack\_weight stays the same, for every instance).

Although greedy ratio algorithm is more efficient, it doesn’t always give the most optimal solution.

**(Graph on the next page)**

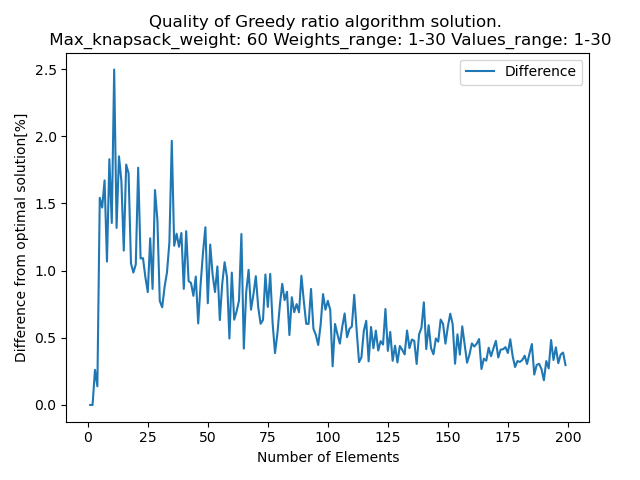


Figure 3: Quality comparison of solution given by greedy ratio algorithm. (Optimal solutions generated employing dynamic approach)

The greedy ratio algorithm sorts the items in decreasing order of value per unit of weight, then it inserts them into the knapsack until it reaches the knapsack capacity. So the algorithm is using heuristics to try to converge to optimal value. The more the elements, the better its approximation becomes, provided that supply of each element is not overly limited. Complexity of greed ratio algorithm is equal to the computational complexity of chosen sorting algorithm as usually O(n) < O(sorting algorithm). So in the case of python implementation, the computational complexity of greedy ratio algorithm is O(nlogn) (Average sorting complexity for python).

Conclusion:

* 1. Knapsack problem is a decision, NP-complete problem, thus there is no known algorithm that can provide optimal solution both correct and polynomial-time in all instances.

Dynamic programing solves the problem in pseudo-polynomial time, so the 0-1 knapsack problem is a weakly NP-complete problem.